NASA Conference Publication 2377

Peer Review of a Formal Verification/Design Proof Methodology

Summary of a sub-working-group meeting
sponsored by NASA Langley Research Center and
held at the Georgia Institute of Technology
Atlanta, Georgia
July 7–8, 1983

NASA
National Aeronautics
and Space Administration

Scientific and Technical
Information Branch

1985

PREFACE

As a part of an ongoing validation research program, NASA Langley sponsored a sub-working-group meeting on the role of formal verification techniques in system validation. This meeting was held at the Georgia Institute of Technology in Atlanta, Georgia, on July 7-8, 1983. The meeting was conducted to assess the value and the state of the art of performance proving for fault-tolerant computers. Particular attention was given to the work done by SRI International concerning the investigation, development and evaluation of performance proving tools.

The two-day meeting began with a review of the techniques developed by SRI to formally verify portions of the NASA LaRC sponsored Software Implemented Fault Tolerant (SIFT) computer design and continued by discussing the soundness of these techniques. Issues concerning the specification language, mappings between hierarchical levels, the interface between the design and code proofs, and SRI's STP design proof environment were addressed.

Approximately 20 leading researchers and system developers were invited to attend this review. These researchers were selected from both recognized advocates of formal verification and acknowledged skeptics.

NASA LaRC's objective in sponsoring this peer review was to examine the technical issues related to proof methodologies. The technical issues discussed are summarized in this report.

PRECEDING PAGE BLANK NOT FILMED

ACKNOWLEDGEMENTS

The organizers of this meeting wish to thank Professor Donald F. Stanat of the University of North Carolina at Chapel Hill for chairing the meeting and Professor Richard DeMillo of Georgia Institute of Technology for hosting the meeting. We also wish to thank Professor Stanat and Teresa A. Thomas for preparing the initial draft of the proceedings. Finally, we wish to particularly acknowledge the time and effort given to this review by the peers themselves.

CONTENTS

PARTICIPANTS

MODERATOR

Dr. Donald F. Stanat
Computer Science Department
University of North Carolina
Chapel Hill, NC 27514
(919) 962-7340

NASA LaRC SIFT CONTRACT MONITOR

Mr. Ricky W.Butler
NASA-LaRC
MS 477
Hampton, VA 23665
(804) 865-3681

LOCAL ARRANGEMENTS COORDINATOR

Dr. Richard DeMillo
Computer Science Department
Georgia Institute of Technology
Atlanta, GA 30332
(404) 894-3180

SRI INTERNATIONAL REPRESENTATIVES

Mr. Peter Michael Melliar-Smith
SRI International
333 Ravenwood Avenue
Menlo Park, CA 94025
(415) 859-2336

Dr. Robert Shostak
SRI International
333 Ravenwood Avenue
Menlo Park, CA 94025
(415) 859-2879

Dr. Richard Schwartz
SRI International
333 Ravenwood Avenue
Menlo Park, CA 94025
(415) 859-5875

RTI REPRESENTATIVES

Mr. James B. Clary
Research Triangle Institute
P.O. Box 12194
Research Triangle Park, NC 27709
(919) 541-7001

Ms. Janet Dunham
Research Triangle Institute
P.O. Box 12194
Research Triangle Park, NC 27709
(919) 541-6562

Ms. Teresa Thomas (Consultant)
Computer Science Department
University of North Carolina
Chapel Hill, NC 27514
(919) 962-2076

PRECEDING PAGE BLANK NOT FILMED

Dr. Jon Bentley
Bell Labs
600 Mountain Avenue
Murray Hill, NJ 07971
(201) 582-2315

Dr. Edward Clarke
Carnegie-Mellon University
Computer Science Department
Schewley Park
Pittsburgh, PA 15213
(412) 578-2628

Dr. Steven Crocker
Aerospace Corporation
Mail Station M1-101
P. O. Box 92957
Los Angeles, CA 90009
(213) 648-5000

Dr. Richard DeMillo
Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332
(404) 894-3180

Dr. Susan Gerhart
Wang Institute
Tying Road
Tyngsboro, MA 01879
(617) 649-9731

Dr. Donald Good
Computer Science Department
University of Texas/Austin
3028 Painter Hall
Austin, TX 78712
(512) 471-1901

Dr. Bret Hartman
Department of Defense
Computer Security Center
Attn: C3
Fort Meade, MD 20755
(301) 859-6380

Dr. Frederick Von Henke
Stanford University
Computer Systems Lab
ERL 403
Stanford, CA 94305
(415) 497-9503

Dr. Richard Lipton
Department EE and CS
Princeton University
Princeton, NJ 08540
(609) 452-3000

Dr. Donald Loveland
Department of Computer Science
Duke University
Durham, NC 22206
(919) 684-3048

Dr. J. McHugh
Research Triangle Institute
P.O. Box 12194
Research Triangle Park, NC 27709
(919) 541-7327

Dr. J. Moore
Computer Science Dept.
University of Texas/Austin
3028 Painter Hall
Austin, TX 78712
(512) 471-7316

Dr. Richard Platek
Oddessey Research Associates
609 West Clinton Street
Ithaca, NY 14850
(607) 277-2020

Dr. Terry Pratt
Dept. of Applied Math & C.S.
Thornton Hall
University of Virginia
Charlottesville, VA 22901
(804) 924-7201

1.0 INTRODUCTION

1.1 Background

Future commercial aircraft will rely heavily on computer-based flight control systems. Failure of a computer system can be tolerated in contemporary commercial aircraft because the flight crew can assume the control functions, but the flight stability margins of future aircraft may be greatly reduced, making human control an inadequate substitute for a computerized control system. For such aircraft designs to be acceptable, the reliability of the flight control systems must be comparable to that of other parts of the aircraft. A reliability criterion that NASA has adopted is that the estimated probability of failure in the flight control system should not exceed 10^{-9} for a flight of 10 hours duration. This level of reliability is higher than is possible with conventional uni-processor hardware and software.

In order to investigate the possibility of attaining such ultra-reliability, NASA-LaRC has sponsored research and development in fault-tolerant computer architectures since 1972. A portion of this effort was directed toward the conception, design, and implementation of a computer system known as SIFT (Software Implemented Fault Tolerance). Since it is not possible to demonstrate by solely testing that a system has a failure rate as small as that proposed for a flight control system, the desired level of reliability must be established by a combination of validation methods including some form of mathematical proof. According to one publication describing SIFT, "The need for such a proof of reliability has been a major influence on the design of SIFT."[1]

Recognizing the importance and the difficulty of a proof of correctness of a computer system such as SIFT, NASA-LaRC awarded a contract to SRI International entitled "Investigation, Development and Evaluation of Performance Proving for Fault-Tolerant Computers." The work done under this contract, dealing with performance proving, is the focus of this report. The objective of the performance proving work was given in the "Statement of Work" of the contract as follows:

> "The objective of this effort is the investigation, development, and evaluation of an experimental facility that will aid in analytically proving the correct performance of embedded fault-tolerant computers in aircraft flight control systems. The end objective of this effort is to provide an automated capability, residing on a host computer, to analytically prove that a design and implementation specification meets the design intent (or is correct)."

1.2 The Formal Verification/Design Proof Peer Review

The Formal Verification/Design Proof Peer Review was conducted to assess the value and the state-of-the-art of design verification for fault-tolerant computers, and to review the work done by SRI International concerning the investigation, development and evaluation of design verification tools. The peer review took place in Atlanta, Georgia, on July 7 and 8, 1983. A list of the members of the panel that conducted the review can be found at the front of this report. The findings of the panel are the subject of this report.

1.3 Objectives of the Peer Review

The objective of the peer review was to assess the role of formal techniques in system validation, with emphasis on the methodology developed by SRI International. A list of issues was formulated prior to the peer review and was reviewed by SRI International prior to the meeting. The list follows:

1. The Role of Formal Techniques in System Validation
   a. To what extent does formal mathematical verification satisfy the goal of total system validation?
   b. When is formal verification more cost-effective than other validation techniques?
   c. What effect does the need for validation have on system design?
   d. Is the approach applicable to large systems?
   e. Can numerical programs, concurrency and other language features be handled?
   f. Does the need for verification affect the likelihood of specification errors?
   g. How is system structure constrained by the need for formal verification?
2. The SRI International Design Proof Methodology
   a. What is the role of the design proof methodology in system validation?
   b. Is the methodology mathematically sound?
   c. Are the components of the methodology (including the specification language, the inter-level mappings, the interface between design and code proof, and the STP design proof environment) individually sound?
   d. Does the approach lead to readable, believable specifications?
   e. Does the methodology make effective use of human and machine resources?

f. Are the axioms which must be assumed in the verification process reasonable? Can they be experimentally validated?

The sequence of questions addressed by the peer review panel (i.e., the Critical Issues of Section 3) was based on this set of issues.

1.4 Sources of Material for this Report

The material contained in this report comes from a variety of sources. At the close of the peer review, each participant was asked to provide written comments on a number of issues; additionally, SRI International representatives were asked to provide a written position statement on the same issues. The position statements and the collective comments of the panel were the principal source of the material contained herein. Additional material in the report is taken from tape recordings and the editors' notes made at the meeting.

1.5 Organization of the Report

The following is a brief description of the contents of the remaining sections of the report.

Section 2 provides background for the SIFT validation work. It also summarizes technical results relative to the issues addressed at the peer review, including the work on the SIFT validation effort.

Section 3 is the body of the report; it is devoted to discussion of a set of "critical issues" discussed at the peer review.

Section 4 examines the issue of claims regarding research accomplishments.

Section 5 summarizes the conclusions of this review.

Appendix A contains a summary of SRI International research relevant to this contract. This summary is an edited version written after the conclusion of the meeting. SRI also wrote the original summary on which this is based.

Appendix B describes the mathematical foundations of the method used in proving SIFT. This work, done by Jose Meseguer of SRI International subsequent to the meeting, was in response to questions of the panel regarding the soundness of the work.

Appendix C, written by SRI International personnel, describes how the material in Appendix B applies to SIFT.

The references are given at the end of the report.

2.0 APPROACH TO VERIFICATION RESEARCH

The SIFT performance proving effort was intended to provide a test of the tools and techniques for the verification of functional properties of fault-tolerant computer systems. The verification effort was coordinated with the SIFT design and development effort. An interesting result of this coordination is that the verification effort uncovered several flaws in the SIFT design, and these discoveries resulted in important design changes.

The work produced a number of other interesting and substantial results concerning the general problem of the design and validation of fault-tolerant computers; these have been reported in the literature in papers dealing with topics such as clock synchronization in the presence of faults, [2] and the Byzantine Generals problem. [3]

The validation of any system requires a characterization of the intended system behavior. In the case of fault tolerant systems, this characterization can be broken into a functional specification that describes the behavior of the system in the absence of debilitating system failure, and an upper bound on the probability of such failure. Given such a system specification, validation of fault tolerant systems can be done by

(1) developing one or more models of the system;

(2) showing that each of the models accurately describes some chosen facet or property of the system and its behavior, and

(3) showing that each model (and hence the system) possesses the desired correctness, temporal or reliability properties.

Of the properties listed in (3) above, "correctness" means that the system correctly computes the desired flight-control functions. The intended "temporal" property is that the system performs all its tasks, including those associated with an operating system as well as the computing of flight-control functions within specified time constraints. The intended "reliability" property can be given as an upper bound on the probability of a system failure that might endanger the aircraft. The performance proving effort was only concerned with establishing correctness; establishing reliability and temporal properties was not part of the effort.

Neither the adequacy of a system specification nor the adequacy of a model's description of a system can be established by any formal means; they are ultimately accepted or rejected on the basis of informed judgment, which may in turn be based on testing or analysis. But showing that a model has a specific property or is satisfied by a more detailed model is, in many cases, a well-formed mathematical problem.

There are three types of models appropriate to a system such as SIFT; these models correspond to the three kinds of properties that the system is to have:

(1) Functional models characterize the functional behavior of the physical components of the SIFT hardware during normal operation (and possibly during some modes of failure). The desired functional behavior of the SIFT system was characterized by means of a set of axioms in a first-order predicate calculus.

(2) Timing models ensure that tasks are performed frequently enough and fast enough to maintain the aircraft in a safe state. These models were not part of the current performance proving project.

(3) Markov models characterize the failure modes and probabilities of the physical components of the SIFT hardware. These provide a basis for showing that the probability of system failure is sufficiently small. These models were also not part of the current performance proving effort.

Proving that a system has a specified functional behavior requires verifying that the programs have the desired properties and that the system hardware correctly implements the programming language semantics. Although some work has been done on characterizing SIFT hardware, this work was not discussed at the peer review. Moreover, no attempt was made to prove the Pascal compiler correct. Thus the work discussed at the peer review assumes the correctness of a Pascal compiler for the variant of Pascal used for most of the coding, and it assumes that the SIFT component machines correctly implemented the target code.

The part of the verification problem, generic to any fault tolerant system, is to show that the system software is functionally correct. The technical approach was to show that the overall design of the system is correct, and verify the code which implements it.

The difficulty of verifying SIFT results largely from the use of several processors in the system. SIFT uses from five to eight identical processors, each with its own clock and its own storage; communication is by a broadcast mechanism. The system software assigns both flight control tasks and operating system tasks to component processors. Fault detection and masking as well as system reconfiguration are managed by software. To guard against failures, tasks are replicated on three or more processors and the task output is determined by majority vote. One task is responsible for detecting permanent failures so that a failed processor's output will be subsequently ignored. The processors maintain a loose synchronization by periodic comparison of the clocks of working processors.

For the functional model, a hierarchy of models was constructed. At the top level of this hierarchy is a set of assertions intended to specify the desired global functional behavior of the SIFT system; these assertions are called the I/O Specifications. The part of the validation effort described at the peer review consisted principally of the design proof effort, which was to show that if the code of SIFT satisfies the appropriate PrePost conditions, then the system implements the top level functional system specifications (the I/O Specifications) so long as a sufficient number of processors are working correctly.

The hierarchy of models between the I/O specifications and the Pascal code reflects many of the design decisions made for SIFT, and proofs involving the hierarchy are referred to as "design proofs." The design proofs are meant to show that each level of the hierarchy (except the top-most one) logically implies the level above it. The lowest level of the hierarchy is a model whose assertions are essentially the code PrePost conditions.

3.0 THE CRITICAL ISSUES

Prior to the peer review, a list of critical issues was formulated and circulated to all participants. A second list of issues was formulated by the panel on the second day of the review. The list of issues in this report was obtained by merging similar issues from these two lists and altering others to better state the issues. Following the meeting, the SRI International representatives were asked to provide a written statement on each of the issues of the original two lists, and the panel members were asked to comment on each issue. In the following, we present each issue and then give SRI International's statement. The statements have been edited to eliminate redundancies.

The first list of critical issues grew out of planning sessions for the review. Contributors to these sessions included representatives of NASA-LaRC, SRI International , Professor Donald F. Stanat, and Professor R. DeMillo. The list was strongly influenced by references in the open literature to SRI International's verification methodology.[4, 5] In many of the following comments and position statements, the phrases "methodology," "verification methodology" and "design verification technique" should generally be regarded as denoting the same concept.

(1) What kinds of verification tasks must be performed to establish the ultrareliability of an aircraft system such as the one that motivated this research? How was this list of tasks constructed?

Response:

If the reliability required (as for SIFT) is such that it cannot be readily confirmed by physical experiment, we are necessarily required to extrapolate the reliability from more easily measured properties, and from a mathematical model of the system. The tasks required for this reliability verification are:

- Definition of the meaning of the term reliability.
- Design of one or more mathematical models cf the system, typically:
  - A stochastic model from which probabilistic reliability measures can be derived, and
  - Functional models describing the behavior of the physical system.
- Measurement of physical parameters of the system required for these models, specifically the rate of occurrence of solid and transient faults, the duration of transient faults, the rate of manifestation of faults as errors, the stability characteristics of the clocks, and the execution times of all the programs.
- Experimental or other verification that the functional models are adequate models of the physical system.
- Verification that the various mathematical models of the system are consistent. In particular, those models that are not physically verifiable must be shown consistent with, or preferably derivable from, the models that are.
- Verification that the various mathematical models imply the specified reliability criterion.

The design verification for SIFT is a demonstration that the specifications (and, in due course, the Pascal implementation), regarded as mathematical objects, have certain mathematical properties. Even were it fully completed, the design verification could not "prove" that SIFT is "reliable." The extent to which our confidence in these physical properties is enhanced by the design verification depends on our confidence that the mathematical models of SIFT reflect the physical reality, a confidence that may derive from experience, physical experimentation, or further mathematical analysis. The value of formal design verification is that it allows us to choose a mathematical model to match the simplest, most easily measured, and most easily modeled aspects of the system, and to deduce more complex, or difficult to measure, properties by rigorous mathematical deduction.

Comments from the peer group:

a. It is important to realize that one does not prove correctness of a system; one proves properties of abstractions (models) of the systems. Therefore, verification cannot guarantee 10^{-9} probability of failure. The biggest problem is finding all the right properties to verify, and this is not a mechanizable process.

b. Modeling the hardware so that the machine operation and machine code can be integrated into the proof scheme is an important and difficult task. The tools that are used must be verified as well; in this case, for example, those would include the STP theorem prover and the Pascal compiler. Because a variant of Pascal is used, and because some of the code is in machine language, the compiler and code verification problem becomes substantial and specific to this application. Resources must also be shown to be adequate -- does everything fit in the available memory? Are all time constraints met? Are numerical computations sufficiently accurate?

(2) What is the most complex system that has been verified using this methodology? Does the approach scale-up to more complex systems?

Response:

The problem of scaling up lies not in the verification methodology or tools so much as in our human understanding and ability to maintain clean designs and specifications for large complex systems, particularly for systems that cannot be fully understood by a single individual. We have no information as to whether the scaling factors are better or worse than those for conventional programming practice.

Comments from the peer group:

a. A small section of the abstract specifications of the operating system routines has been verified. There is no reason why, in principle, the hierarchical decomposition approach cannot be applied to more complex systems. Scaling up appears to be a problem of implementation rather than theory.

b. The SRI methodology may be almost at the limit of its utility in the partial design proof that was done. The verification system is based on a form of the predicate calculus, and this may be too weak a modeling tool. While any system, in principle, could be modeled, in fact it is too hard to model systems of any realistic complexity. The methodology needs much higher level primitives that relate to actual programming constructs (e.g., data objects, program-level operations and control structures) before it begins to be applicable to real systems.

(3) Does the methodology handle numerical programs, concurrency, and other language features? Is it appropriate to abstract concurrency out of the code?

Response:

Yes, the methodology does handle a limited form of concurrency. The design verification of SIFT involved a proof that a set of asynchronously interacting computers maintains consistency and cooperates reliably. The proof was performed by brute force techniques, involving the explicit representation of time.

A very important aspect of the verification of the correct design of the asynchronous interaction was the structuring of that interaction into a stylized form. It is our opinion that code verification in the presence of unrestricted asynchronous interaction, even though possible in principle, is so difficult that it will never be of practical importance. Successful verification of concurrent systems will always depend on well-structured communication between processes. Such structure allows separate consideration of the behavior of program sections within processes and of the interaction between processes.

The validity of that separation of considerations requires a metaproof. This metaproof was beyond the capabilities of STP but our new specification and verification has sufficient expressiveness and deductive power to undertake such proofs.

It is our opinion that the structuring of communication between asynchronous processes is not required merely for verification. It is also good programming practice in all contexts, and is necessary to obtain a reliable and maintainable program.

In the SIFT design, there is no communication during process execution. Design proof is used to verify that the schedules ensure that all input data are available prior to and throughout process execution. This allows us, at the lower levels of the proof, in particular for the code proof, to consider each process execution as an atomic, uninterruptible action. The decision was necessary to keep the mechanical verification within the state-of-the-art, but we believe that it was an appropriate decision for SIFT on other grounds, too.

New techniques are now becoming available that will greatly reduce the cost and difficulty of specification and verification of asynchronous systems, and we expect to incorporate one of them, *interval logic,* (an extension of conventional temporal logic) in our new specification and verification system. We intend to evaluate the use of *interval logic* by attempting to verify the correctness of the highly asynchronous clock synchronization algorithm of SIFT (only 35 lines of Pascal declarations and statements, but probably beyond the ability of any current mechanical verifier).

The problems of verifying numerical programs are mathematical problems about what properties should be specified for such programs, and about how to analyze the numerical algorithms so as to prove such properties. This is an important area for future research in verification although, unfortunately, it has received scant attention.

Other language features can complicate verification, such as exception handling. Programs that make use of such a feature may well be difficult to verify. It is our opinion, however, that equivalent programs, with the same functionality achieved by use of elementary program constructs and without use of the higher level language feature, will be even more difficult to verify. Again, it is the required functionality of the program that must be verified and is difficult to verify. The availability of a structured language feature makes the task easier rather than more difficult.

Comments from the peer group:

a. Some aspects of arithmetic are handled, but for the most part, numerical programs, concurrency, and interrupts are not handled. These are among the issues that must be addressed. A major weakness of the methodology is the small effort made to relate what is proved with STP to the requirements of the actual program under consideration. It is not clear whether the methodology can be extended to handle these features.

b. Abstracting concurrency out was necessary with the present state of the art, but it isn't adequate for a real concurrent system. Ultimately, this issue must be addressed. Concurrent programs are notorious for very subtle timing errors which cannot be easily handled by exhaustive testing.

(4) How does the methodology handle errors such as those in hardware, design, axiomatic specification, programs, and maps between levels?

Response:

The design verification technique cannot address faults in either the highest level specifications (the requirements) or in the lowest level specifications (the machine). The probability of error in the highest level specifications is reduced to the extent that those specifications are short and amenable to human inspection.

The design verification technique, together with code verification, can only demonstrate that the specifications and the code, regarded as mathematical objects, have certain mathematical properties. The demonstration of appropriate properties may increase our confidence that the specifications, and code, reflect our intent. Even fully completed, the design verification cannot "prove" that a system is "reliable".

Comments from the peer group:

a. It is somewhat premature to assess how well errors are handled. The notion of levels of specification is a good one, but inevitably specifications will change, and it would be best if the impact of such change were constrained to those areas necessarily affected.

b. Part of our difficulty in addressing this issue may result from not understanding what an error is. At the lowest level errors can occur in hardware design or in modeling of the hardware. Errors in specification at the highest level are errors in judgment; we can only ask that the methodology gracefully handle changes at that level. As for errors in the lower level specifications and the mappings between levels, we are not sure what is meant, since we have not been given a careful specification of what properties the lower specifications and mappings must have. Thus, until the methodology is carefully defined, what constitutes an error will remain unclear.

c. The system depends on a social process to find errors in the specification and design, a process which appears not to have worked well in this case.

(5) What quality factors are addressed or affected by the methodology (e.g., testability, reliability, maintainability, performance)?

Response:

The methodology does not directly address any of these quality factors. Rather, it is the quality of the design that affects the quality factors. To the extent that a quality factor can be formally defined as a mathematical property, the methodology can be used to increase our confidence in that property of the design. We know of no reason why verification should adversely affect any quality factor.

The design verification technique draws attention to the algorithms of the design and away from immediate implementation issues. Thus verification may have a substantial indirect effect on other quality factors.

We see no reason why other structural approaches included to facilitate testing or maintainability should be precluded by the need to verify. Indeed we envisage that any verified system is likely also to be subjected to very thorough testing. We note that formal verification can be used to justify programs that are hard to test, for example programs containing asynchronous interactions.

It is to be hoped that future verification systems will be able to verify non-trivial performance characteristics of systems. Such verification may be able to identify potential performance problems that are difficult to identify by other means. Verification of non-trivial performance properties is still a research area, however.

Formal verification does strongly and directly enhance one useful quality, that of portability. Formally verified programs do not depend on any undocumented nonstandard feature of the system on which they are implemented, and their dependence on documented special features is explicit, greatly facilitating portability.

Comments from the peer group:

a. Performance is often adversely affected by the program structure required for verification, and I would expect such an effect from this methodology or any other. It is essential that performance requirements be included from the start, and that largely automatic reverification of modified code be possible.

b. Formal verification does not preclude the use of other techniques to insure such things as testability and maintainability. Moreover, there is potential for improving testability and maintainability by following the hierarchical design and verification approach.

c. Ideally, the design methodology will lead to similarly structured system design. This can be expected to improve modifiability.

d. The methodology probably does not introduce limitations, but more complex systems will be harder to specify and verify. This may be a substantial problem because some kinds of complexity, such as robust code, are desirable but difficult to characterize in a way that fits in with formal verification.

e. Verification of a system may lead to undue confidence that it will work correctly. One possible effect would be similar to that of not putting very many lifeboats on an "unsinkable" ship; since you know that nothing can go wrong, there is no need to guard against disaster.

(6) What are the constraints on system structure imposed by the design methodology? How does the methodology influence system design and implementation? Can an implementation be based on the hierarchical design?

Response:

To a first approximation, the design verification methodology should impose no constraints on the system structure. To the extent that the system is clean, elegant and simple, the design verification is easier and quicker. If the requirements necessitate a more complex system, the design verification will be correspondingly more complex and expensive. Gratuitous complexity in the system, leading to unnecessarily difficult design verifications, will have to be avoided.

The design verification technique encourages, even mandates, clean elegant orthogonal designs. We do not regard this as a constraint. The design characteristics that facilitate verification are characteristics that are recommended practice for all systems. Probably they are even more critical for systems that must depend on incomplete testing and informal justifications.

It is of the essence of a hierarchical design approach that the more abstract levels of the design can be quite far removed from the implementation. The lowest levels of the design specifications are however very close to the code that implements them; they express, in specification language style, exactly the same manipulations that the implementing code expresses in imperative style. Implementation is thus trivial, as it should be, for the aim of a design specification hierarchy is to capture the design decisions in the specifications.

At any level in the hierarchy, one can provide an implementation of the abstract machine that meets .the axiomatic specification. But the more abstract levels of the hierarchy, obviously, provide only an abstract, partial definition of an implementation. At a lower level, where the design is more complete, the implementation is more clearly defined. The PrePost level in the SIFT hierarchy is particularly well-suited to mapping to an immediate implementation in Pascal -- the data structures and operations are directly mapped to Pascal data structures and procedures. In fact, we produced a Pascal implementation of the code from the PrePost specification, although the code on the system delivered to NASA LaRC was, unfortunately, developed prior to the availability of the PrePost specification.

Comments from the peer group:

a. Top-down design is encouraged by this methodology; each routine must be small and simple. The critical question, then, is whether simplicity will prevent robustness or even basic functionality. Perhaps only a trivial system could meet the constraints imposed by this methodology.

b. The structure of the design proof does not have to be reflected in the final design. Factoring to simplify proof need not correspond to factoring to simplify implementation.

c. It seems that this approach separates the design of a system from the implementation so well that the two have nothing to do with each other. Thus, while one could try to use the hierarchical design as a guide for implementation, this might be ill-advised or even impossible. It is difficult to address this question better without a careful characterization of the constraints that characterize an admissible hierarchical design.

d. A number of changes to the design occurred between the original description [1] (e.g., 1978 SIFT) and the current version. In the current version,

i. priority-based preemptive periodic scheduling is replaced with a static pre-planned schedule,

ii. task lengths, which were arbitrary in original design, must fit in a fixed length time slot,

iii. dynamic allocation of tasks to processors is replaced by a static assignment, and

iv. the complexity of the application programmer's task is substantially increased.

It is not clear why these changes were necessary, but it is likely that a number of them can be attributed to the constraints imposed by the methodology. That does not necessarily render them undesirable; we'd rather have a reliable awkward system than an unreliable pretty one, but it is probably foolish to think that the methodology does not constrain the design.

(7) How do you know axioms are correct? Are the axioms understandable?

Response:

The hierarchical technique allows a very succinct top level specification of the requirements on the system. It is our belief that this facilitates human inspection of the specifications and increases our confidence that they do indeed reflect our true needs.

The design verification technique, together with code verification, can only demonstrate that the specifications, the code, and the mappings, regarded as mathematical objects, have certain mathematical properties. The demonstration of appropriate properties may increase our confidence that the specifications, and code, reflect our intent. The extent to which our confidence in these physical properties is enhanced by the design verification depends on our confidence that the mathematical models reflect the physical reality, a confidence that may derive from experience, physical experimentation, or further mathematical analysis. The value of formal design verification is that it allows us to choose a mathematical model to match the simplest, most easily measured, and most easily modeled aspects of the system, and to deduce more complex, or difficult to measure, properties by rigorous mathematical deduction.

Comments from the peer group:

a. The top level is simple enough for a flight engineer to understand, but this does not mean that he can see that the processing system is correct. The methodology could be defined more tightly (to exclude mapping functions) so that self-containment of the top level specifications is guaranteed.

b. Considerable expertise is required to understand the axioms. Maybe their theorem prover encouraged a clumsy, hard-to-read notation. It is a serious shortcoming of the methodology that the axioms are hard to understand at the top level and get harder to understand as one descends to lower levels. It took one and one half days for a panel of experts to understand the top level. This speaks for itself.

c. SRI International's design verification is impressive, even if it cannot be understood by those untrained in logic.

d. The top level appears to characterize the system adequately, but it needs work.

e. The axioms that characterize component hardware must not only be "correct", but they should also be complete and consistent. They should describe all states of the machine including failure rates. The axioms require extensive testing to make them sufficiently believable. One can write axioms that make it possible to prove the theorems required, but there is no guarantee that the machine actually makes the lowest level axioms true at all times.

f. SRI International chose to axiomatize not only the top level, but also some nonfundamental concepts that are used in various parts of the proof. It was shown at the peer review that three such axiomatizations of conventional mathematical concepts (MOD.AXIOM, SEQ.EQUALITY.AXIOM, and SET.SELECTION in Section 8 of Investigation, Development, and Evaluation of Performance Proving for Fault-Tolerant Computers, Interim Report) [6] are either inconsistent or incorrect in that they did not correspond to the conventional notions and that the deviation was apparently not intentional. This demonstrates that axiomatic descriptions of basic concepts can lead to errors. An axiomatic description of a large system will be difficult to understand in any case, and is very likely to be inconsistent in some ways. Unnecessary axiomatization of basic concepts is likely to compound the problem.

g. The discovery of three bugs in the axioms would be disturbing if we seriously expected this system to be a production verification system, but has no direct bearing on the work at this stage.

(8) How are changes in the specification or in the code handled?

Response:

During a design verification (or code verification), it is necessary to make many minor changes to formulas and code. Such changes affect other portions of the proof, which must then be corrected and repeated. It was found to be critically necessary that the verification system automatically keep track of changes and guard the soundness of the evolving proof without imposing undue restrictions on the order in which the proof is developed. The result of the verification process is a proof that can be examined and reproduced independently of how it is arrived at.

Comments from the peer group:

a. Minor code changes are probably handled well. Major algorithm changes would require changes in the axioms at all levels. Reverification will be difficult and costly.

b. The verified version already differs from the delivered version, and the latter is undergoing change. The decoupling of design and code proofs from the running code is a serious problem. It is impossible to speak of "SIFT as a verified system" unless a connection is established. Furthermore, changes after that will require reverification. Will they ever resist changing (even "trivially") a verified system to handle some convenience or "improved approach"?

(9) What skills are necessary to use the methodology?

Response:

A user of the design verification technique and STP must have an understanding of mathematical logic approximating a typical undergraduate course in that topic. No understanding of the internal mechanisms of STP is required.

The use of design verification does not in any way reduce the amount of experience with, and understanding of, the application that is required of the designer. The quality of the design and the feasibility of design verification are entirely dependent on the skill of the designer.

Comments from the peer group:

a. Significant mathematical sophistication seems to be required of both the designer and the application programmer. It is not clear to what extent the application of formal methods will ever become easy and commonplace, i.e., easy to use by those untrained in logic. Familiarity with the particular application is probably less important than familiarity with formal methods.

(10) Does the methodology facilitate the apportioning of validation effort according to the level of risk involved for different kinds of errors?

Response:

The selection of properties to be specified as requirement on the system by the top level specifications may reflect an assessment of the properties that are most important. Similarly, the selection of the lowest level, below which verification is not carried, may also reflect an assessment of risk. For example, a decision has to be made as to whether the lowest level of the verification should extend to axiomatic specifications only, to the Pascal code, to the binary machine code, to the microprogram, to chip level logic design, to the component level design, or even into analog properties of semiconductors.

However, once decisions have been made as to the top-level and bottom-level specifications, the hierarchical design verification technique regards all properties as equal in importance, and is equally rigorous and effective against all errors that fall within its scope.

Comments from the peer group:

a. Ideally, it would be possible to concentrate on particular properties so as to deal with them more rigorously than with others.

(11) How does the methodology constrain the end-user?

Response:

To a first approximation, the design verification methodology should impose no constraints on the end-user or his system. One must distinguish between constraints imposed by the SIFT design and constraints imposed by the needs of formal verification; we address the latter.

An application program, like any other program, is expected to satisfy a functional specification and to comply with an axiom concerning scheduling frames. It may also be required to comply with resource limitations (time, space, use of shared resources, etc.). If the program interacts with other programs, it will have to obey a protocol specification.

Comments from the peer group:

a. It is not clear what restrictions on the application programs are implied by the design proof. These restrictions should be stated in the requirements documents.

b. The methodology forces simplicity of design and may do so to an unreasonable degree. Non-preemptive scheduling (all tasks must fit in a fixed time slot) is a constraint that apparently evolved from application of the methodology, and one cannot know how that will affect the programmer until some application programs are written.

c. If the application programmer is considered to be the end-user, then the SIFT experience would suggest that he may be affected negatively. Compared to the description in "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," [1] the present system, according to Butler [7] requires much more of the application programmer. He must write his programs so that they all execute within a specified fixed period. He must assign tasks to processors and build schedule tables, and he must do so for every possible combination of working and faulty processors. He must also build voting tables that correspond to each precalculated schedule table. Each of these additional responsibilities makes the application programmer's task harder, and it makes it harder to ensure that he's done his job correctly.

(12) Does the verification methodology apply to the application software?

Response:

The SIFT design does not tolerate errors in the application software. To the extent that the correct operation of the application programs is necessary for the reliable operation of the system, the application programmer must verify his programs, or use other techniques to establish that his programs are sufficiently reliable.

Comments from the peer group:

a. Correctness of the total system depends on correctness of the application programs, but confidence may be enhanced by partial verification. Verifying only the kernel routines is worthwhile, even if the application programs are not subjected to the same rigorous review.

b. Clearly, the applications must meet the same reliability criteria as the rest of the system if total reliability is to be preserved.

(13) To what extent does formal mathematical verification address the goal of total system validation? To what extent does mechanization of proofs contribute to the verification of programs and confidence in the programs?

Response:

Design verification is merely a demonstration that the specifications (and the code), regarded as mathematical objects, have certain properties. The extent to which the verification process addresses the actual physical behavior of the system depends on the degree of confidence that these mathematical objects reflect the physical reality. This confidence may derive from experience, physical experimentation, or further mathematical analysis. The value of formal design verification is that it allows us to choose a mathematical model to match the simplest, most easily measured, and most easily modeled aspects of the system, and to deduce more complex, or difficult to measure, properties by rigorous mathematical deduction.

The extent to which design verification can address a total system depends only on the extent to which that total system can be modeled mathematically. Clearly, the more of the system that is to be modeled, the more complex the mathematical model becomes and the more difficult deduction within the model becomes. It is clear that the techniques applied to a portion of SIFT address only a restricted aspect of that total system. As new and improved verification systems become available, investigation of more complete systems will become possible.

It has been our experience that mechanical verification is essential if we are to have the highest level of confidence in our proofs, and thus in our programs. During the design verification effort, so many minor faults were found in the SIFT specifications that we would now have little confidence in any proof that has not been subjected to mechanical checking.

During the various phases of the design verification, manual and mechanical, we also found four faults in the design that required changes to the implementation. These design faults all related to inadequate fault tolerance in special circumstances. Even with the benefit of knowledge of the design fault, we are unable to envision any test sequence that could have been expected to expose the design fault. One can have no confidence that programs subjected to testing alone do not contain subtle design faults such as those.

But, however complete and rigorous a formal verification is, the properties verified are only properties of a mathematical model. Formal verification necessarily makes an assumption about the relationships between that model and the real system, relationships that can and should be investigated.

Other forms of validation, such as program testing, make different assumptions from those of formal verification. It is our opinion that those assumptions are far less likely to be justifiable than the assumptions made for formal design verification, but it is very important that they are different. Consequently, it is possible to use testing to investigate and substantiate the claimed relationships between the formal model and the actual system. Thus testing remains an essential complement to verification in raising our level of confidence in our programs and systems.

Comments from the peer group:

a. The SIFT verification effort definitely shows that a combination of existing theory, methodology, technology, expertise, and review can produce proofs of properties about abstractions of the underlying system.

b. There is no real alternative to formal verification for this kind of reliability. Mechanization of proofs is vital for large proofs and detects errors in proofs more reliably than inspection by humans. The value of verification is twofold. First, it is a totally independent approach to finding errors from "walk-throughs" and other code-scanning techniques. Second, the rigor of having to reconcile the implementation axiomatization with the "performance" constraints imposed by the next higher level axioms via generating proofs forces fresh looks at various aspects of the design and implementation.

c. I feel strongly that formal modeling of the total system, formal verification of that model, and direct, automatic generation of code from the formal model are required if an overall system is to be reliable. Ad hoc, informal methods are hopeless. The complexity of a system like SIFT suggests that mechanization of large parts of the verification is essential, regardless of the methodology used. Thus, formal modeling and automatic verification aids are essential.

   However, the technical approach taken is too low-level in its modeling abilities and is therefore hopelessly inadequate for actual systems. But the potential of formal verification for SIFT-like systems should not be judged on the basis of this effort.

(14) What other validation activities should be used in conjunction with this methodology? Are they compatible?

Response:

We know of four general approaches to improving the reliability of designs (typically but not exclusively software designs): software fault tolerance techniques (such as recovery blocks and N-version programming, but also including other less structured techniques), testing in a simulated environment, testing in actual operation, and verification of various degrees of formality. These techniques are by no means mutually exclusive.

The inclusion of software fault tolerance code into a program, aside from increasing its size and complexity, introduces two special verification problems. If code is introduced that can be reached only through a test detecting some form of error condition, it is possible that the verification may fail to check that code, the required conditions being established by demonstrating that the test must "always" fail. Similar problems arise in testing such code. Consequently, redundant code, if ever exercised, may be found to be very unreliable. The second problem is that heuristic repairs to data structures may be very difficult to justify, formally or even informally.

The Recovery Block technique does not suffer seriously from these two problems and appears to us to be quite compatible with design and code verification.

Comments from the peer group:

a. Compatibility with other activities is probably feasible, but, to the knowledge of the panel, it has not been investigated.

b. Any others that are available can be used.

c. Extensive testing and high-level programming languages with clear control structures are necessary.

d. It is assumed that a number of tasks are performed perfectly by the designers and implementors. These tasks include (1) specification of the system, (2) construction of a failure model, (3) verification of the code, and (4) construction (and therefore performance) of the verification system. These assumptions invite disaster because we are not generally able to perform the tasks perfectly. It appears that the assumptions made awkward the use of techniques that would be helpful when an assumption fails. Maintainability and robustness were not goals in the SIFT work, and any system built with this methodology would suffer as a consequence.

(15) Are design proofs in the absence of code proofs relevant? Do design proofs interface well with the code verification?

Response:

Each level in the SIFT specification hierarchy is the specification of an abstract machine. Using incremental design proof, each level is shown successively to be an implementation of the next higher level in the hierarchy. In our view, the "code" is merely one level in this hierarchy. Clearly, the closer that the physical implementation is to the level of abstract machine that has been proven to have certain properties, the more credible the claims about the "correctness" of the system actually executing. In this context, of course, the Pascal code is running on an abstract Pascal machine, itself running on a BDX930 machine, etc. The lower levels must be shown or assumed to be consistent with the physical implementation. The decision as to the appropriate level of design detail to carry the proof is really a question of cost-effectiveness.

The traditional verification condition generation paradigm bridging the gap between code and specification obscures much of the insight on the part of the designer as to why the code meets its specification. We are currently working on modifying this paradigm to be able to reason about properties of abstract and concrete programs in the same manner – eliminating the currently awkward interface.

Comments from the peer group:

a. The design proofs produce the PrePost conditions to be used in the code proofs; thus they play a vital part in preparing for code verification. The work in this area is very impressive.

b. If the design proof is well done, it should simplify construction of the code proof. However, the usefulness of the SIFT design proof is suspect in view of how difficult the axioms are to understand and the fact that the code proof was never carried out correctly.

c. Design proofs in isolation from the code and coding proofs are worthless. The design proofs must eventually be reconciled with the code, and this task appears to be impossibly difficult.

4.0 PREVIOUSLY REPORTED RESEARCH ACCOMPLISHMENTS

Scientific workers are expected to describe their accomplishments in a way that will not mislead or misinform. Members of the peer review panel felt that many publications and conference presentations of the SRI International verification work have not accurately presented the accomplishments of the project; several panel members, as a result of the peer review, felt that much of what they thought had been done had indeed not been done. Because of these misunderstandings, a substantial part of the panel's effort was devoted to trying to establish what had been accomplished. Several panel members expressed serious concerns regarding both the possible effects of such misunderstandings on those seeking to apply the work and the reactions that might follow the discovery that the work had been overstated.

The research claims that the panel considered to be unjustified are primarily in two categories; the first concerns the methodology purportedly used by SRI International to validate SIFT, and the second concerns the degree to which the validation had actually been done.

Many publications and conference presentations concerning SIFT appear to have misrepresented the accomplishments of the project. In many cases, the misrepresentation stems from omission of facts rather than from inclusion of falsehood. The following quotations are illustrative.

> "This paper describes the methodology employed to demonstrate rigorously that the SIFT computer meets its reliability requirements." [4]

> "The formal proof, that a SIFT system in a 'safe' state operates correctly despite the presence of arbitrary faults, has been completed all the way from the most abstract specification to the PASCAL program. This proof has been performed using STP, a new specification and verification system, designed and developed at SRI International. Except where explicitly indicated, every specification exhibited in this paper has been mechanically proven consistent with the requirements

on SIFT and with the Pascal implementation. The proof remains to be completed that the SIFT executive performs an appropriate, safe, and timely reconfiguration in the presence of faults." [5]

Quotes such as these misled the panel into believing that the methodology and proof of SIFT were in a far more mature state than was seen at the peer review.

5.0 CONCLUSIONS

The purpose of the peer review was to evaluate the role of formal verification techniques in system validation, focusing on the verification part of the SIFT work.

In general, SRI believes that the SRI proof effort, in its incomplete and only recently formalized state,* represents a substantial contribution to specification theory. Clearly, the verification process is only as good as the specification of the system. To address this problem, SRI utilized a parameterized hierarchy of specifications. With this hierarchy, it can be guaranteed that any behavior observable from the simple top-level specifications will in fact follow from the lowest detailed level specification.

Whether other systems or methodologies could address similar issues, SRI believes that the SRI effort goes far in actually accomplishing this. This is in contrast to demonstration of a collection of correctness properties expressed and proved at the level of the code. In many cases, the specifications submitted for the user's scrutiny are as long as and more difficult than the actual code.

Clearly, this effort has not been completed. The hierarchy has not yet been carried down to the actual Pascal Code, assembly language, or transistor level. The more elementary the components of the bottom level in the hierarchy, the more believable the proof. It would be a mistake, however, to discount the SRI proof effort because the hierarchy has not yet been carried down to a "running implementation" level.

The incompleteness of the SIFT verification exercise caused concern at the peer review. Many panel members who expected (from the literature) a more extensive proof were disillusioned. It was the consensus of the panel that SRI's accomplishment claims were strongly misleading. The panel members felt it was the responsibility of researchers not only to reveal the full potential of their work (as SRI did) but also to carefully define the status of the work, clearly indicating

* Subsequent to the peer review, SRI generated a formal description of their methodology which appears in Appendix B.

what had and had not yet been done.

The methodology, as it was described at the peer review and in the documentation provided, was not specified in a way that would provide substantial help in validating a system. It placed no constraints on the form of the axioms or how the formal logic would be used. It did not specify what constitutes a 'mapping' between layers, leaving open such matters as consistency, completeness and form of these mappings.* But perhaps the state of the art (or at least of this project) is best revealed by the fact that work (described in Appendix B) intended to address some of the panel's concerns led SRI workers to conclude that their proof was incomplete in a way they had not previously perceived (see Appendix C).

Members of the review panel were doubtful that the hierarchical approach used in the design proofs can be classified as a methodology, although the approach taken by SRI to validate SIFT is novel and notable. They had serious doubt about the claim that the correctness of the system design can be judged by examining only easily understood high-level functional specifications. The claim falters on two counts. First, the difficulty encountered by members of the panel in understanding the top-level assertions of the SIFT operating system makes it seem likely that inadequacies or errors in the specification could easily be overlooked, even by someone familiar with the application. Second, some members of the panel were concerned that, even though the understanding of the top-level assertions might be perfect, underlying models in the hierarchy could be constructed in ways that logically imply the levels above while betraying the intended model.

In summary, a meaningful formal proof methodology was not clearly described to the peer reviewers. Consequently, there was doubt expressed as to the existence of a detailed methodology. Subsequent work submitted by SRI International in Appendices B and C may support the claim of a feasible methodology, but this work was not available to the panel.

The review panel, because of preoccupation with establishing the existence of a SRI formal proof methodology, never properly addressed the role of formal techniques in total system validation and the state-of-the-art of formal proof methodology development.

* The methodology description in Appendix B clarifies some of the questions raised at the review but was unavailable during the review process.

APPENDIX A

OVERVIEW OF SRI ACCOMPLISHMENTS

SRI International
Menlo Park, California

The purpose of the "Performance Proving" project, under which SIFT design verification efforts were performed, was to advance the state of the art in verification, rather than to prove SIFT, just as the objective of the companion SIFT project was to advance the state of the art in fault tolerant architecture rather than to build a computer. Thus, some of what was done was valuable science but not directly relevant to the SIFT design verification itself; some was investigations that failed or were subsequently evaluating an experimental facility that could possibly be used to aid in analytically proving the correct performance of embedded fault-tolerant computers in aircraft flight control systems.

The specific tasks in our statement of work were:

(1) To extend the performance proof technology to handle systems, and specifically handle fault-tolerant computers and flight control systems.

(2) To develop an experimental proof facility to automate the proofs.

(3) To demonstrate the technology and facility by design proving the SIFT fault-tolerant computer.*

When we started the work, we were only vaguely aware of the difficult technical issues that had to be faced to verify a real-time (and "real") operating system, let alone real application programs. Among the difficult technical issues that we quickly became aware of are:

- how to express easily understood requirements for complex systems,
- how to handle concurrency that is driven by hardware interrupts and not easily expressible using a high-level programming language,
- how to verify the lower levels of the system (assembly code and hardware logic),
- how to model the environment in which the computer is embedded (e.g., the "real" state of the aircraft and the computer hardware, which is not captured by any observable variable), and
- how to verify performance properties that vitally impact the correctness of a system (e.g., will the application programs terminate within their allotted time and run within their allocated space).

Early in the project we identified the need for what we subsequently called design proof. Our ultimate aim was to use as a requirement the Markov reliability model. As it became clear that we would not be able to prove anything about failure rates or about the distribution of the time interval from the occurrence of a fault to the instant at which errors show up to a voter, we decided to abandon

* Although the task statement only indicated "design" proof, it was our objective to carry out a verification of running code. Furthermore, at the beginning of the project we did not fully appreciate the value of design proof.

this approach. Instead, we placed our effort on a functional requirement, which culminated in the I/O model. Subsequently, we introduced the three lower level models, each of which captures a related set of key decisions; for example, the Replication Model axiomation task assignment and replication, and voting.

Regarding the verification of code, we had hoped to separately consider the Pascal portion of the operating system and that portion which had to be written in assembly code. We did not want to consider any approach based on verifying the compiler. In the end, we never made much progress on the assembly code. Moreover, the Pascal used for the delivered version SIFT involved some hooks into the hardware which we did not formalize. (For example, overloading of the assignment statement was used to indicate broadcasting of a buffer to all processors.) Hence, a cleaned up version of the Pascal code was constructed. The code we did consider for verification was close to the Pascal portion of the delivered SIFT system, and handled most of the features that were addressed in the design verification, but was not the running executive.

Our initial overall approach, in retrospect naively selected, was to use HDM as the basis for describing a system. In principle HDM as it existed at the beginning of the project was suitable, being capable of describing hierarchically-constructed programs and models, having a specification language (SPECIAL) that supports abstraction and is linkable to existing theorem provers, being able to support high level programming languages, and allowing for different kinds of implementations (e.g., combinations of hardware, assembly language and high level programming language). In anticipation of HDM being suitable, we started to develop a verification system based on HDM and the Boyer/Moore theorem prover (see below).

However, as we started to specify and verify SIFT we became aware of certain deficiencies of the approach. HDM, requiring that each operation be atomic, did not fully support concurrency. Before we could extend the HDM model of computation to allow for concurrent operations, it was necessary for us to fully understand what was required to specify and verify concurrent programs. The Boyer/Moore work described below was a first step in this direction which, unfortunately, did not lead to an extension of HDM in time. This prevented us from producing a mechanical verification of the theorem that relates to the noninterference of the SIFT processors on each other – at least as related to the verification.

In addition, we uncovered some deficiencies in the SPECIAL language that led to longer and clumsier than necessary specifications. For example, SPECIAL does not support parameterized modules meaning, for example, that a SET of INTEGER and SET of WORD cannot be described with a single specification that is instantiated. Moreover, SPECIAL requires model-theoretic specifications (complete specifications), while several of the more abstract models in the SIFT design hierarchy seemed to be better specified with property-theoretic specifications (incomplete specifications). We also experienced some difficulty in using the Boyer/Moore theorem prover, particularly for the design proofs. As described below, we were not able to effectively generate lemmas from which the Boyer/Moore theorem prover could generate the proofs. STP provides parameterized types, the ability (through quantifiers and other features) to produce property-theoretic specifications, and a level of man-machine interaction that we found better suited to human-assisted proof.

The SIFT design verification, manual as well as mechanical, has produced many interesting results, among them:

- interactive consistency and Byzantine agreement,
- reliable clock synchronization algorithms,
- the demonstration (still continuing) of the correspondence between the states and transitions of a Markov reliability model and a functional model of a system,
- the hierarchical design proof technique,
- the STP verifier.

The work undertaken under task (1) was:

The development of a method for verifying numerical algorithms (Milton W. Green). The method factors the proof into steps: verification assuming infinite precision followed by verification of precision properties. We illustrated the method on the King-Floyd exponentiation program, proving it mechanically using the Boyer/Moore theorem prover. We hope this work will provide us with the technology required to verify application programs whose specification involves an accuracy requirement.

The verification of a very simple control system (Robert S. Boyer, Milton W. Green and J. Moore). The control system assures that a vehicle subject to wind disturbances remains on course. The specification is interesting in that it includes that state of the vehicle which is not captured by any observable variable. A mechanical verification was carried out using the Boyer/Moore theorem prover. It was our hope that this work would inspire others to tackle the specification and verification of real flight control systems.

The verification of hardware logic (Robert E. Shostak). The method developed was analogous to that of Floyd for software. Basic components, such as gates, flip-flops, and registers were axiomatized and hand proofs were developed for a frequency comparator. A mechanical proof of the Muller C-element, a circuit that interfaces asynchronous elements, was also carried out. It was hoped that this work would be the basis for verifying the hardware logic of a real computer, but any serious attempts in this direction are in the future.

The formal definition of computer architectures (Robert S. Boyer and J Moore). A formal definition was developed in the Computational Logic for the Bendix BDX930 processor. This work was important in showing that the instruction set of a real processor could be formally defined in a logic supported by a theorem prover. In addition, some inconsistencies in the informal definition were uncovered. Unfortunately, the definition was just too big to be useful.

The work performed under task (2) consisted of:

The formal definition of an HDM subset (Robert S. Boyer and J. Moore). This definition describes the meaning of an HDM verification of a SPECIAL module. This very elegant definition was to be used as the basis for arguing about the soundness of the HDM-Pascal verification system, which was eventually abandoned.

Continued development of the Boyer/Moore theorem prover (Robert S. Boyer and J. Moore). The majority of the support for this development came from ONR and NSF. The NASA project supported the extension of the theorem prover to negative numbers.

The Meta-VCG (Mark S. Moriconi and Richard L. Schwartz). This tool generates verification conditions from a program and a specification, much as a conventional VCG. The Meta-VCG is however language independent and accepts as input a Hoare sentence definition of the programming language semantics.

The HDM Pascal Verification System (Dwight Hare). This is a conventional VCG type program verification system, using a VCG derived from the Meta-VCG and accepting SPECIAL specifications and Pascal programs. It was initially designed to interface to the Boyer/Moore theorem prover, but was modified at the last moment to interface to STP. Unfortunately, this modification was not fully completed, thus requiring manual intervention into the verification conditions before they can be proved.

The STP Verification System (Robert E. Shostak, Richard L. Schwartz, and P. M. Melliar-Smith). This system is based on decision procedures developed under an AFOSR contract by Robert Shostak, extended to provide a strongly typed first order logic with schemes and the beginnings of a man-machine symbiosis for proof development.

The work performed under task (3) was:

Development of abstract reliability models for SIFT (Robert E. Shostak and Leslie B. Lamport). This was our first attempt to develop an abstract statement of the functional properties of SIFT and relate to a model close to the operating system.

Development of design specifications for SIFT using SPECIAL (P. M. Melliar-Smith). This produced a formal definition of the Lamport and Shostak models expressed in SPECIAL. For the reasons described above and reflecting the inadequacies of SPECIAL, this work was abandoned.

The proof of the dispatching code for the SIFT executive (Robert S. Boyer and J S. Moore). This work aimed to prove that a dispatcher, written in BDX930 machine instructions, would handle interrupts and invoke the execution of application tasks. Ultimately, the aim was to demonstrate coordination of six asynchronous processors at the machine instruction level. The task failed, in part because the formal definition of the BDX930 completely filled the store of the theorem proving environment, and in part because we did not develop an appropriate specification for the requirements on a dispatcher. In commencing this task, we had hoped to provide a link from the real dispatcher (the workhorse of SIFT) to our design proof. In retrospect, the verification of the dispatcher might have been feasible if carried out with respect to the specification that the constraints imposed by the scheduler preclude task interference.

Design proof of SIFT using the Boyer/Moore theorem prover (Milton W. Green and J S. Moore). This work started with a Pascal "implementation" of the I/O model, which enabled us to test the specification with real data. Following this, we wrote the I/O and Replication models in the Boyer/Moore theory. The verification attempt did not succeed because of the difficulties we had in generating the proofs in the Boyer/Moore theory. However, it should be noted that J. Moore did succeed in replicating the STP proof of correspondence between the two top level models of the SIFT design hierarchy, using the "axiom" and "proof-check" features at the Boyer/Moore theorem prover.

The top-level specifications of SIFT, the I/O specifications, are best expressed as very abstract property theoretic specifications which attempt to express only the essential requirements on the system for it to behave reliably. Such specifications, we believe, are necessarily partial and leave much free to be defined as the design is elaborated. But the computational logic, based on recursive function theory, requires total, well founded definitions of all functions. This results in the specification of much auxiliary mechanism and yields a specification that is much more specific than is necessary or desirable.

The Boyer/Moore theorem prover attempts to be a fully automatic theorem prover, capable of proving theorems without human interaction, and as such it is probably the most advanced in the world. This automatic capability is provided by sophisticated, and very complex, heuristics. Unfortunately, even with these heuristics, the theorem prover is not able to prove non-trivial theorems without human assistance, provided in the form of lemmas. Unfortunately the human assistance has to predict the needs of the heuristics that will generate the automatic proof. The normal user does not have the detailed understanding of the internal structure of the prover and its heuristics needed to provide this assistance.

The design proof of SIFT using STP (P. M. Melliar-Smith and Richard L. Schwartz). This proof was started towards the end of the project, and even now is not yet complete. It was however successful in demonstrating the feasibility of design proofs of non-trivial properties of SIFT. In particular, the proof verifies that a SIFT system in a safe state will execute the scheduled tasks, provide them with correct inputs and mask processor faults in their results. The proof does not demonstrate correct or timely reconfiguration, nor does it connect the design to the Markov model on which reliability predictions are based. SRI International is currently being funded by NASA-LaRC to continue the proof using new tools, with greater rigor, and to extend it to cover reconfiguration, interactive consistency and clock synchronization. We would also hope to incorporate the Markov model into the model hierarchy.

The verification of the Pascal code sections derived from the SIFT design specifications (Dwight Hare). As indicated above, these programs and their proofs do not involve any arguments about concurrency. We proved properties about sequential programs, relying on the design proof to assure us that the programs did not interfere with each other. This, too, was started too late in the project and after the Pascal verification system had been abruptly converted from one prover to another. SRI found the STP style of automated proof to be more effective than that of the Boyer/Moore theorem prover in carrying out the design verification of SIFT, although we confess to being somewhat biased. In particular, the quantification feature of STP allowed us to more easily write incomplete specifications, and the parameterization feature led to reasonably general theories. Furthermore, we found the style of interaction afforded by STP, although often tedious, to be more natural. On the other hand, skilled users of the Boyer/Moore theorem prover appreciate its extensive heuristics. The Pascal code sections verified were only those required to support the incomplete design proof, and was not the code of the running SIFT. In particular, the global executive, reconfiguration, interactive consistency, and clock synchronization were not verified.

APPENDIX B

FOUNDATIONS FOR DESIGN PROOF METHODOLOGY*

* A result of Peer Review; provided by Jose Meseguer, SRI International, to further explain methodology.

The purpose of this appendix is to make explicit the mathematical basis of the method employed in the proof done on SIFT so that its soundness, generality, and what mathematically can be accomplished by its use can be more clearly perceived by everybody.

The goal of the proof approach is, of course, to show that an executable implementation at the bottom level of the hierarchy satisfies all the properties asserted in the top level specification. In this way, a system can be understood in terms of the simpler high level specification without reference to the details of its low level implementation. The specifications themselves need not be executable; what is being shown is that any executable bottom level implementation can be viewed as a high level abstract machine.

1.1 Preliminaries

A **many-sorted first order signature (with equality)** is a triple $\Sigma = (S,F,\Pi)$, where S is a set of **sorts** (denoted s, s′, etc.), F is a set of **function symbols** that comes with an "arity" function ar:F→(S•xS) (its elements are denoted by f, g, etc., and if ar(f)=(s1...sn,s) by f: s1...sn→s), and Π a set of **predicate symbols** also with an arity function ar: P→S^+ (elements are denoted by P, Q, etc.) and such that for each sort s in S there is a distinguished equality predicate symbol $=_s$, i.e., ar($=_s$)=ss.

The **first order language** of Σ, denoted L(Σ), is defined inductively in the usual way: first terms with variables (denoted t, t′, etc.) are built using F, then atomic formulas are built from terms using Π, and then formulas (denoted A, B, etc.) are built from atomic formulas by means of the usual logical connectives.

A first order theory (with equality) is a pair T=(Σ,Γ) where Γ is a set of formulas in L(Σ) (called the **axioms** of the theory). By the language of T, L(T) we just mean the language of its signature, L(Σ). The theory T=(Σ,Γ) is called S-**sorted** if the set of sorts of Σ is S. A **model** M of an S-sorted first order theory T=(Σ,Γ) is an S-sorted sequence of sets M=$M_{s\in S}$ together with: (i) for each f: s1...sn→s in F a function f_M:M_{s1}x...xM_{sn}→M_s; (ii) for each P in Π (other than the equality predicate symbols) with, say ar(P)=s1...sn, a function P_M: M_{s1}x...xM_{sn}→ {**true,false**} and such that M **satisfies** all the axioms in Γ. (The notion of satisfaction is the obvious one, i.e., interpreting the equality predicate symbols as the ordinary equality predicate, each term t in L(T) denotes a function t_M from sorts of M to a sort of M, and each formula A denotes a truth-valued function A_M from sorts of M; we then say that M **satisfies** a formula A, denoted M |= A, iff A_M has the constant value **true**.) The class of all the models of a theory T will be denoted **Mod**(T.) Given a theory *T*, if a formula A in L(T) is satisfied by all the models of T we will write T |= A, and say that A is

a semantic consequence of T.

By a **syntactic consequence** of a theory T we mean a formula A such that there is a finite subset Γ_0 of Γ such that A can be deduced from Γ_0 by the usual rules of many sorted first order logic with equality; we then write T |- A. By the completeness theorem for first order logic we know that a formula is valid in all models of a theory if and only if it is derivable from the axioms, i.e.,

T |= A if and only if T|-A

1.2 Views

It is very common to have situations in which two different logical theories are naturally related. Whenever there is a relationship between theories, a corresponding relationship is induced between their models; for instance, if a theory T' extends or enriches the language and the axioms of T (i.e., T' has perhaps additional new function and predicate symbols and possibly additional axioms) then the inclusion $T \subseteq T'$ provides a *view* of T' from T, and allows us to *view* or regard each model of T' as a model of T, by just forgetting about the additional function and predicate symbols. For instance, the inclusion of the theory of semigroups into the theory of groups allows us to view each group as a semigroup. In other common situations. however, the relationship between two theories is more involved than just a simple inclusion; for instance, every boolean algebra can be viewed as a boolean ring, and this corresponds to a view of the theory of boolean algebras from the theory of boolean rings in which the ring operations are expressed as derived expressions from the basic boolean operations of union, intersection, and complement. In logic textbooks such as Shoenfield's *Mathematical Logic,*[8] Section 4.7, views are called "interpretations"; the term "view" is due to Goguen and Burstall, who treat views in full generality in their recent work on "institutions" [9] and show that their basic properties do not depend on the kind of logic chosen: equational, first order, ..., etc. In this Section we explain the basic facts about views for first order logic; this will provide the basis for our discussion of the SIFT hierarchical specification and verification in Section 4.

Definition 1: Given an S-sorted theory $T=(\Sigma,\Gamma)$ and an S'-sorted theory $T' =(\Sigma,\Gamma)$ a **view** of T' from T is a pair V=(v,V), where $v: S \to S'$ is a function and $V: (F \bigcup \Pi) \to L(T')$ is a function mapping each function symbol f of Σ to a term V(f) in L(T'), and each equality predicate symbol $=_s$ to $=_{v(s)}$, and any other predicate symbol P of Σ to a formula V(P) in L(T') in such a way that:

1. "Arity is preserved", i.e. if $f: s1...sn \to s$ in F, then the term V(f) will denote a function $V(f)_M: M_{v(s1)} x...x M_{v(sn)} \to M_{v(s)}$ in each model M of T', and similarly if P in Π has ar(P)=s1...sn, then V(P) will denote a function

$V(P)_M: M_{v(s1)} x ... x M_{v(sn)} \rightarrow$ **{true,false}** in each model M of T'; note that the map V can be extended by induction to a function $V: L(T) \rightarrow L(T')$, i.e., we can think of V as a translation from the language of T into that of T'.

2. "The translation of the axioms is provable", i.e., for any axiom A in Γ we have

$$T' \vdash V(A)$$

We denote such a view by V: T→T'. Given a view V: T→T', each T' model M can be viewed as a T model V*(M), with $V*(M)_s = M_{v(s)}$, $f_{V*(M)} = V(f)_M$, and $P_{V*(M)} = V(P)_M$. It is clear that V*(M) satisfies the axioms Γ, since for any such axiom A we have $A_{V*(M)} = V(A)_M$, and $M \models V(A)$ (thus $V*(M) \models A$), since $T' \vdash V(A)$. In other words, V* is a well defined function

$$V*: \mathbf{Mod}(T') \rightarrow \mathbf{Mod}(T)$$

By the completeness theorem we know that a theory is consistent if and only if it has a model. Thus from the definition of view above we obtain:

Corollary 2. If V: T→T' is a view and T' is consistent, then T is consistent.

A very important property of views is that they are "theorem preserving". We give a simple proof of this result using the completeness theorem. The same result for a quite restricted notion of view is the "Interpretation Theorem" in Shoenfield's text, [8] pp. 62-63; for a fully general treatment allowing different logics see Goguen & Burstall.[9]

Theorem 3: For V: T→T' a view and A a formula in L(T), if $T \vdash A$ then $T' \vdash V(A)$.

Proof: By the completeness theorem, $T \vdash A$ if and only if $T \models A$, i.e., $M \models A$ for each M in **Mod**(T). In particular, we then have $V*(M') \models A$ for each V*(M') in V*(**Mod**(T'), i.e., $M' \models V(A)$ for each M' in **Mod**(T'), i.e., $T' \vdash V(A)$, again by the completeness theorem, q.e.d.

A very useful corollary of this theorem is that views can be composed to yield new views, i.e., they form a "category."

Corollary 4: If (v,V): T→T' and (v', V'): T'→T'' are views, then (v'ov,V' oV):T→T' ' is also a view.

Proof: For each axiom A in Γ of T we have $T' \vdash V(A)$ by hypothesis, and then

$T'' \vdash V'$ (V(A)) by the above theorem, q.e.d.

1.3 The Proof of Correctness in SIFT

The notion of view just given provides a foundation for the method of hierarchical specification and verification employed in the proof of SIFT.

The authors specified the SIFT system in a "hierarchical way" and this means precisely that they provided a sequence of first order theories $T_1,...,T_n$ (T_1 the "top level" and T_n the "bottom level" specifications) and for each level a view of the next level below, i.e., views V_i: $T_i \rightarrow T_{i+1}$, $1 \leq i \leq n-1$. The proof consisted precisely in showing mechanically that each of these V_i was indeed a view, i.e, that

$$T_{i+1} \vdash V_i (A) \text{ for each axiom } A \text{ in } T_i$$

This formalizes the method used in the proof. Although the axioms corresponding to each T_i are grouped together in the report, the report itself does not make explicit enough what the signatures and the views are; STP did not provide mechanical support for hierarchies at the time when the proof was done. This forced all the levels to be lumped together into a single unstructured theory.

Several important consequences follow immediately from the above observation and the results on views proved in the previous section, namely:

1. Since views are closed under composition, any model M of the bottom level theory T_n (i.e., any T_n "machine") becomes, once the correctness of the views has been established, a model of the top level theory T_1, i.e. a "correct" T_1 "virtual machine", namely $V_1*(...(V_n*(M)...)$.
2. Once the views have been proved, consistency of all the theories $T_1,...,T_{n-1}$ follows **for free** from the consistency of the bottom level theory T_n, and that in turn follows from exhibiting a T_n machine, i.e., an "implementation," by the completeness theorem.
3. Theorems of the top level, simplest theory, follow automatically as theorems of the lowest level implementation, by Theorem 3.

1.4 Realizability

In SIFT the task execution and voting schedules are parameters of the system. Provided certain constraints on the schedule are met, the system is expected to execute accordingly. This brings about a further concern besides correctness, namely the *realizability* of any possible schedule by an implementation at any level. A parameterized proof of this property seems best, and that can be formalized by making explicit an input theory T_{input}, whose models are schedules, together with views J_i:$T_{input} \rightarrow T_i$ at each level of abstraction that compose nicely with the views that relate the levels of the hierarchy. As we shall see the proof of the "realizability property" can be pushed down to the bottom level specification for which one has to show that any "input models" satisfying the specifications for the inputs (scheduling and reliability properties) can be extended to a T_n model, where T_n denotes the specification of the bottom level. After showing that the views from T_{input} compose nicely, it will then follow by the correctness property of the preceding section that any input model could be extended to an implementation at *any* level in the hierarchy. It is my understanding that a full proof of realizability has not been carried out for SIFT to date, but that plausible informal arguments can be given in support of the realizability property. The purpose of this Section is to explain what the realizability property means in terms of views, and what remains to be done to have a full proof.

The main observation is that in the SIFT design the function and predicate symbols F_i and Π_i in the specification T_i of the i-th level can be partitioned into **input** and **output** disjoint subsets, i.e., $F_i = F_{i_i} \bigcup F_{i_o}$ and $\Pi_i = \Pi_{i_i} \bigcup \Pi_{i_o}$, and that there is an additional theory T_{input} (corresponding to the "schedule axioms" in Levitt's *Investigation, Development, and Evaluation of Performance Proving for Fault-Tolerant Computers.)* [10] An explicit account of the partitions at each level has been given in the section "A Guide to the SIFT Hierarchy." A view J_n: $T_{input} \rightarrow T_n$ (n the bottom level) is implicitly given in Levitt's report, since the "schedule axioms" are part of the axioms at the bottom level. The realizability property means that for each T_{input} model M there is a T_n model M′ such that

$$J_n*(M')=M$$

or in other words, that the view J_n:$T_{input} \rightarrow T_n$ has the following "realizability property,"

Definition 5: A view V:$T \rightarrow T'$ has the **realizability** property if and only if the function $V*$: **Mod** (T') $\rightarrow$**Mod**(T) is surjective.

Let us assume that the realizability property had already been established for J_n:$T_{input} \rightarrow T_n$ and that additional views J_i:$T_{input} \rightarrow T_i$, $1 \leq i \leq n-1$, are provided and we show that the following diagram of views commutes

$$\begin{tikzcd}
 & & (T_{input}) \arrow[dll] \arrow[dl] \arrow[dr] \arrow[drr] & & \\
T_1 \arrow[r, "V_1"] & T_2 & \cdots & T_{n-1} \arrow[r, "V_{n-1}"] & T_n
\end{tikzcd}$$

Then the realizability property would follow trivially for each of the $J_i{:}T_{input} \rightarrow T_i$, $1 \leq i \leq n-1$, and in particular for the view of the top level J_1. Also, each specification level would then be "self contained" in the sense that a model of the system could be understood at that level by knowledge of only the input model that it realizes and the specification at that level; knowledge of the other levels would not be needed.

APPENDIX C

TUTORIAL BY SRI INTERNATIONAL TO ILLUMINATE PROOF METHODS

The foundational material in Appendix B by J. Meseguer defines the basis for the SIFT proof methodology. In this appendix we describe the application of that basis to the proof that was actually performed for SIFT.

The goal of the design proof approach is, of course, to show that an executable implementation at the bottom level of the specification hierarchy satisfies all the properties asserted in the top level specification. In this way, a system can be understood in terms of the simpler high level specification without reference to the details of its low level implementation. The specifications themselves need not be executable; what is being shown is that any executable bottom level implementation can be viewed as a high level abstract machine.

As discussed by Meseguer, each view in the SIFT hierarchy represents a theory of SIFT. Each view includes an image of the schedule table and reliability input functions and defines SIFT at that level of abstraction. An image of the schedule table constraints must appear at each level, in terms of the scheduling abstraction at that level. Mappings between levels map each input and output function and predicate symbol to symbols from the level below.

Given this, by the correctness and realization results given by Meseguer, one obtains a specification of a self-contained, parameterized SIFT "machine" at each level. One is guaranteed that any properties of the top-level machine will follow for the lowest level machine, as shown by Theorem 3 that the views are "theorem preserving". One is assured that the behavior described by the I/O Specification is indeed behavior that will be exhibited by each of the lower level machines.

The consistency result, Corollary 2, shows that if the bottom level of the specification hierarchy is consistent and views are established and proved between each adjacent pair of levels, then each one of those specifications is consistent. Note that the correctness property ensures, in effect, that the mappings from lower levels have been implemented to form a correct view to the higher level.

If the full proof of correctness and realizability is completed, only curiosity necessitates scrutiny of lower level views. Consider for example ON.DURING, a function that was discussed at the peer review. I/O Specification axioms IO.A2 and IO.A5 describe a certain relationship between ON.DURING and RESULT. In effect, for a task and an iteration, IO.A2 requires that if ON.DURING is true (and other conditions are satisfied) then RESULT has a certain value (the "correct" value). Similarly, IO.A5 requires that if ON.DURING is false, and if any other tasks examines that value, RESULT will be the bottom (null) value. ON.DURING is thus an input, available for the user to define, that "switches tasks on and off", a very simple abstraction of a schedule.

It is our contention, based on Meseguer's work, that no understanding of any lower level specification is necessary to understand this specification. Further, given the completed proof, any interpretation that can be given to ON.DURING from this I/O Specification is a valid interpretation in each of the lower level models too.

The I/O Specification contained no constraints on the user's choice of ON.DURING. This resulted from our failure at that time to appreciate the Realizability result of J. Meseguer, a result that we believe is new and unpublished. The effect of the omission is equivalent to an undertaking that SIFT is capable of executing all of its tasks simultaneously. More properly, the schedule constraints expressed at lower levels should have been mapped into equivalent, more abstract constraints in the I/O Specification.

Reviewing the proof of SIFT, as it was performed and presented at the peer review, in the light of the work of J. Meseguer, it is clear that the proof is incomplete; the relatively simple demonstrations required for the Realizability property were omitted. But we are satisfied that the proofs that were done were correct and appropriate, and represent the greatest part of the complete proof.

Here we provide a guide to the structural components of the actual proof. The components parallel those discussed by Meseguer. We have not completed the mappings of the schedule table views at each level, and thus we give only the lowest level view of the schedule constraints. This work must of course be done before the proof is said to be complete.

The following pages describe the major function symbols comprising the input and output function symbols of each view in the hierarchy. We have omitted auxiliary (derived) function symbols and several functions related to schedule table bounds, task constants, etc. Where we cite actual page numbers for reference, page numbers refer to the Final Report on the Sift Proof. [10]

Several comments on the completeness of the formal method of hierarchical proof for the actual SIFT proof conclude the guide.

I/O LEVEL

Axioms: pp. 109-113
Major Axioms: IO.A2, IO.A5
Final Status of IO.A2: page 289
Final Status of IO.A5: page 291

Major Primitive Function and Predicate Symbols

	Inputs	Mapping From Next Level
Schedule Structure	inputs	identity
	beg	identity
	end	identity
	of	identity
	real.time	identity
Task Function	apply	identity
	function	identity
Scheduling	ON.DURING	RP.D7
Reliability	TASK.SAFE	RP.D9A

	Outputs	Mapping From Next Level
Results	result	RP.D6

REPLICATION LEVEL

Axioms: pp.125-134
Major Axioms: RP.A3, RP.A2, RP.D4
Mappings: with axioms
Lemmas and Proofs: pp. 137-160

Major Primitive Function and Predicate Symbols

	Inputs	Mapping From Next Level
Schedule Structure	inputs	identity
	beg	identity
	end	identity
	of	identity
	real.time	identity
Task Function	apply	identity
	function	identity
Scheduling	POLL.FOR.OF	BR.RE.MAPPING.4
Reliability	SAFE	RE.BR.MAPPING.9

	Outputs	Mapping From Next Level
Results	ON	BR.RE.MAPPING.5
	ON.IN	BR.RE.MAPPING.6
	IN	BR.RE.MAPPING.7, BR.RE.MAPPING.8

ACTIVITY LEVEL

Axioms: pp. 171-180
Major Axioms: BR.A41, BR.A9C, BR.A6B, BR.A9A, BR.A35, WITHIN.SKEW
Mappings: pp. 183-185
Lemmas and Proofs: pp. 189-242

Major Primitive Function and Predicate Symbols

	Inputs	Mapping From Next Level
Schedule Structure	INPUTS	PP.MAPPING.9
	beg	identity
	end	identity
	of	identity
	real.time	identity
	start	identity
	finish	identity
Task Function	function	PP.MAPPING.8
Scheduling	sched	PP.MAPPING.3, PP.MAPPING.4
	pollby.for	PP.MAPPING.12
Reliability	working.during	identity

	Outputs	Mapping From Next Level
Results	inputin.of	PP.MAPPING.7
	datafilein.for.on	PP.MAPPING.5, PP.MAPPING.6

PREPOST LEVEL

Axioms: pp. 245-253
Major Axioms: EXECUTE.ACTIVITY, VOTE.ACTIVITY
Mappings: pp. 257-259
Lemmas and Proofs: pp. 263-282

Major Primitive Function and Predicate Symbols

	Inputs
Schedule Structure	p.inputs p.config
Task Function	task_results
Scheduling	sched_table real_to_virt
Reliability	working.during
	Outputs
Results	input datafile

One can see from the actual proof transcript that the structure of the axioms, mappings, and proofs correspond to the foundation for the hierarchical proof described by Jose Meseguer.

We do point out, however, that only the correctness portion of the hierarchical proof has been completed. The input theory of schedules and reliability, given on pages 163-167, has yet to be formally mapped to the hierarchy of views. Schedule constraints are currently stated in terms of the input function symbols of the Activity-level specification. It remains to map these constraints up to the input function symbols of the I/O level.

Two informal extensions to the methodology were needed to handle the realities of the multiprocess program proofs and the issue of the reliability input function symbols (WORKING, SAFE, etc).

First, to ensure the meaningfulness of the treatment of the reliability functions denoting the schedule of processor failure, it is necessary that these symbols NOT map to implemented functions in the lowest level of the SIFT system. That is, a SIFT Executive implementation, could, without additional constraints, be correct because it is deciding who to believe during voting by taking advantage of these functions, presumably provided by underlying hardware! This would result in an implementation that is correct but useless in that it must be run on a machine that could not itself be mechanized. The methodology, as formalized, does not take this into account; it is an interesting topic for future research.

Second, strictly speaking, the lowest level view in the hierarchy is not the SIFT Executive, but the multi-process system consisting of n communicating SIFT Executive processes.

The PrePost specification level and the code proof below it address the question of a single iteration of the Executive operating on a working Pascal machine, treated as an ATOMIC, non-interruptible, action. That each iteration of each instance of the Executive is called by a hardware clock interrupt is completely beyond the scope of the proof. The significance of this proof to the formalized notion of correctness given earlier is certainly not immediate.

A supplementary argument is required to reconcile this. A major portion of this argument was mechanized as part of the verification effort, and constitutes a large part of the overall SIFT proof. Further metatheoretic argument is required to formally complete the justification.

Briefly, the idea is the following... Each iteration of the SIFT Executive performs the activities it believes have been scheduled for the current time period (subframe). Recall that each process is operating asynchronously, attempting by periodic clock task resynchronization to remain within some maximum skew of the other clocks. We wish to prove that despite the degree of process asynchrony present, that:

> Whenever an iteration of the Executive on a working processor, according to its clock, begins to execute a scheduled Execute or Vote activity, the corresponding input for that activity will have arrived, and will remain stable throughout the execution of the activity, provided the input process is executed on a working processor.

In particular, voted input values for a Vote activity and broadcast output values from an Execute activity must be present and stable during the time the values can be used.

By establishing this, we can consider activities to be effectively atomic, and employ classical input/output sequential verification techniques to establish properties of the Executive that will be valid even in a multi-process configuration.

This is not a new technique -- the concept is that of Non-Interference, introduced by Owicki and Gries [11] to deal with asynchronous process interaction where, despite the asynchrony, there is not a high degree of process interaction. This is deliberately the case in the SIFT design: activity schedules are statically determined, with scheduling constraints to specifically guarantee that no contention for data will occur. Without factoring the multiprocess proof into separate steps of (1) establishing non-interference, and (2) proving that the behavior of each effectively atomic activity is correct, we do not believe the proof could have even begun to reach the level of completion that we have achieved.

A portion of the non-interference justification has been completed. Activity level lemmas BR.LEMMA.17 and BR.LEMMA.17X prove that, due to scheduling constraints, an Execute activity on a safe processor will result in other safe processors having the result of the Execute, according to local time, at the beginning of the following subframe. The status of this proof is found on page 293. It follows from scheduling constraints and an assumption (BR.LEMMA.FOR.LES.TO.PROVE) that any process with a safe clock task will be WITHIN SKEW of other clocks. See page 180 for the statement of the assumptions. BR.LEMMA.FOR.LES.TO.PROVE has not been mechanically verified; see Lamport and Melliar-Smith [12] for a hand proof. An inductive proof of BR.LEMMA.4 then establishes that the value present in all safe processors at the beginning of the subframe following the Execute will be that used for later voting. Lemmas BR.LEMMA.48 and BR.LEMMA.49 then establish that no other Execute or Vote activities can interfere with these executed and voted results.

The main Execute axiom of the Activity level, BR.A41, stating that results of an Execute activity will be in all other working processors according to the appropriate local time, is actually proven in two different ways: once from the PrePost and Pascal levels (see page 280), discounting any issue of timing, and once from BR.LEMMA.17 (see page 293), on the basis of the timing argument. Following up from these two proofs of the Activity level axiom are two proofs for IO.A2. Statuses for these are given on pages 297 and 289 respectively.

The applicability of this argument to the actual SIFT system depends on several additional assumptions. Each iteration of the Executive, the Execute activity for each application task, and each Vote activity MUST complete in its allotted time. It is this that J. Moore's proofs of the BDX930 interrupt handling begin to address. Furthermore, execution of an application task must have no effect on the state other than to perform a broadcast of its results to other processors.

As J. Moore notes in his section of the SIFT Final Report [10] , it is currently beyond the state of the art to formally prove that machine level interrupt handling will actually ensure the Executive iteration properties on which our proof ultimately depends. We concur, but do not believe that this in any way detracts from the significance of what we HAVE proved. It simply suggests areas for further research.

REFERENCES

1. J. Wensley, et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proc. IEEE*, vol. 66, no. 10 , October 1978.
2. L. Lamport and P.M. Melliar-Smith, *Synchronizing Clocks in the Presence of Faults* , SRI International , February 1982.
3. L. Lamport, et al., "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, July 1982.
4. P. M. Melliar-Smith and Richard Schwartz, "Hierarchical Specification of the SIFT Fault-tolerant Flight Control System," CSL-123, SRI International, March 1981.
5. P.M. Melliar-Smith and Richard L. Schwartz, "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System," *IEEE Transactions on Computers*, vol. C-31, no. 7, July 1982.
6. K.N. Levitt, et al., *Investigation, Development, and Evaluation of Performance Proving for Fault-Tolerant Computers, Interim Report*, June 1983.
7. R.W. Butler, "An Assessment of the Real-Time Application Capabilities of the SIFT Computer System," NASA Technical Memorandum 84482, NASA, April 1982.
8. J. Shoenfield, *Mathematical Logic*, Addison-Wesley, 1967.
9. J. Goguen and R. Burstall, "Introducing Institutions," *Proceedings of the Logics of Programs Workshop, CMU*, 1983.
10. K.N. Levitt, et al., "Investigation, Development, and Evaluation of Performance Proving for Fault-Tolerant Computers," NASA Contractor Report 166008, NASA, August 1983.
11. S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Communications of the ACM*, vol. 19, no. 5, May 1976.
12. P.M. Melliar-Smith and L. Lamport, "Synchronizing Clocks in the Presence of Faults," *Journal of the ACM*, vol. 32, no. 1, January 1985

1. Report No.
NASA CP-2377

2. Government Accession No.

3. Recipient's Catalog No.

4. Title and Subtitle
PEER REVIEW OF A FORMAL VERIFICATION/DESIGN PROOF METHODOLOGY

5. Report Date
June 1985

6. Performing Organization Code
505-34-13-33

7. Author(s)

8. Performing Organization Report No.
L-15992

9. Performing Organization Name and Address
Research Triangle Institute
Center for Digital Systems Research
Research Triangle Park, NC 27709

and

NASA Langley Research Center
Hampton, VA 23665

10. Work Unit No.

11. Contract or Grant No.

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Washington, DC 20546

13. Type of Report and Period Covered
Conference Publication

14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

As part of an ongoing validation research program, NASA Langley Research Center sponsored a sub-working-group meeting on the role of formal verification techniques in system validation. The meeting was held at the Georgia Institute of Technology, Atlanta, GA, on July 7-8, 1983. The meeting was conducted to assess the value and the state of the art of performance proving for fault-tolerant computers. Particular attention was given to the work done by SRI International concerning the investigation, development, and evaluation of performance proving tools. Approximately 20 leading researchers and system developers were invited to attend this review. These researchers were selected from both recognized advocates of formal verification and acknowledged skeptics. NASA Langley Research Center's objective in sponsoring this peer review was to examine the technical issues related to proof methodologies. The technical issues discussed are summarized in this report.

17. Key Words (Suggested by Author(s))
Verification
Design proof
Methodology
Formal proof
Software engineering
Software

18. Distribution Statement
Unclassified - Unlimited

Subject category 61

19. Security Classif. (of this report)
Unclassified

20. Security Classif. (of this page)
Unclassified

21. No. of Pages
55

22. Price
A04

For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA-Langley, 1985

Formal proof
Software engineering

Subject category 61

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 55	22. Price A04

For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA-Langley, 1985